Lightweight Annotations for Controlling Sharing in Concurrent Data Structures

Zachary Ryan Anderson
David Gay
Mayur Naik

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-44
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-44.html

March 29, 2009

Copyright 2009, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Lightweight Annotations for Controlling Sharing in Concurrent Data Structures

Zachary Anderson
University of California, Berkeley
zra@cs.berkeley.edu

David Gay Mayur Naik
Intel Research, Berkeley
{david.e.gay,mayur.naik}@intel.com

Abstract

SharC is a recently developed system for checking data-sharing in multithreaded programs. Programmers specify sharing rules (read-only, protected by a lock, etc.) for individual objects, and the SharC compiler enforces these rules using static and dynamic checks. Violations of these rules indicate unintended data sharing, which is the underlying cause of harmful data-races. Additionally, SharC allows programmers to change the sharing rules for a specific object using a *sharing cast*, to capture the fact that sharing rules for an object often change during the object's lifetime. SharC was successfully applied to a number of multi-threaded C programs.

However, many programs are not readily checkable using SharC because their sharing rules, and changes to sharing rules, effectively apply to whole data structures rather than to individual objects. We have developed a system called *Shoal* to address this shortcoming. In addition to the sharing rules and sharing cast of SharC, our system includes a new concept that we call *groups*. A group is a collection of objects all having the same sharing mode. Each group has a distinguished member called the *group leader*. When the sharing mode of the group leader changes by way of a sharing cast, the sharing mode of all members of the group also changes. This operation is made sound by maintaining the invariant that at the point of a sharing cast, the only external pointer into the group is the pointer to the group leader. The addition of groups allows checking safe concurrency at the level of data structures rather than at the level of individual objects.

We demonstrate the necessity and practicality of groups by applying Shoal to a wide range of concurrent C programs (the largest approaching a million lines of code). In all benchmarks groups entail low annotation burden and no significant additional performance overhead.

1. Introduction

The increasing prevalence of multicore processors requires languages and tools that support safe concurrent programming. C itself provides no such support — most concurrency errors in C programs happen silently, and do not become observable until the program fails catastrophically. In these cases, it is common for the bug to be difficult to deduce from the way the program fails.

A number of tools [32, 11] have been developed for modern languages like C# and Java to detect one type of concurrency error: the data-race. A data-race occurs when two threads access the same location in memory without synchronization, and at least one of the accesses is a write. In these systems an exception is raised immediately whenever a data-race occurs. The fail-fast approach used by these runtimes is in agreement with good systems design. However, in the same way that a program crash is a symptom of some deeper problem, a data-race is a symptom of unintended data sharing.

SharC [1] is a recently-developed system that attempts to identify instances of unintended data sharing. Using SharC, programmers make type annotations that describe how data is intended to be shared (or not) among threads. These annotations are called *sharing modes*. Additionally, SharC provides a checked cast operation called a *sharing cast* that a programmer can use to identify places in a program where the sharing mode for an object changes. Using a combination of static and dynamic analysis, SharC checks that no data is shared among threads in the program except in ways that have been explicitly declared by the programmer. This stronger condition also precludes the existence of data-races.

Unfortunately, SharC has a fairly serious limitation that prevents it from being easily applied to some widely-used multithreaded C programs: sharing casts apply to individual objects to which only a single reference remains. Therefore, changing the sharing mode of data structures, such as linked lists and trees, is either unsafe, impossible, or prohibitively expensive in SharC. Complex data structures occur frequently in all C programs including multithreaded C programs. Some of the structures used have different sharing modes at different points in time. For example, data structures are initialized before being shared with other threads, or become private to a thread after being removed from shared queues, lists, and hash tables. Applying SharC to such programs is thus difficult or impossible.

We have extended SharC to address this shortcoming. In addition to the existing sharing modes and sharing cast of SharC, we have developed a new concept that we call *groups* by borrowing ideas from region, ownership, and dependent type systems. For the purposes of this paper, we use the name *Shoal* to refer to SharC extended with groups, and SharC to refer to the original system. A *group* is a collection of objects all having the same sharing mode. Each group has a distinguished member called the *group leader*. When the sharing mode of the group leader changes by way of a sharing cast, the sharing mode of all members of the group also changes. We distinguish these casts from SharC's single-object casts by calling them *group casts*. We ensure soundness of group casts by requiring that all external pointers into a group have a type that *depends* on a pointer to the group leader, and using reference counting to prevent casts when more than one pointer to the group leader remains. Objects can be added or removed from groups, and group casts can be used to combine groups. We present the syntax along with examples of group use in Section 2, and formalize and prove the soundness of our group type system in Section 3. Our formalism for groups is not strongly tied to the concept of a sharing mode, and could be used to track other properties, such as tainting, immutability, etc. We hope that groups and group casts represent a promising approach to describing the runtime evolution of such properties.

We have used Shoal, to check data sharing in several interesting benchmarks, including the GIMP image manipulation program which is over 900k lines. Our benchmarks include scientific codes using graphs and trees to organize calculations; a simple webserver using an in-memory cache shared among threads that process requests from clients; and two image manipulation programs that use threads to improve performance and hide latency. To support these benchmarks, Shoal includes a sharing mode that checks a common idiom in parallel scientific applications; barrier synchronization is used to separate updates in one thread from reads by other threads. We have observed that Shoal's annotation burden is manageable. For small (a few thousand line) programs we made about one change for every 50 lines, but for the GIMP we only needed one for every 10 000 lines. Additionally, making the right annotations requires minimal program understanding — the GIMP required one person only three days to annotate. The scientific benchmarks had overheads around 40% (mostly due to concurrent reference counting), the overheads on the other benchmarks were less than 20%, and overheads were not affected by the number of threads. We discuss our experimental results in full in Section 4.

To summarize, this paper makes the following contributions:

- We present *groups* and *group casts*, lightweight mechanisms for describing data structure properties, and evolution of these propreties. Groups borrows ideas from region, ownership and dependent type systems to describe sets of objects with a common property, represented by a distinguished leader.
- We describe how groups can be used to specify the sharing strategy used in concurrent programs that use complex data structures.
- We demonstrate the practicality of Shoal by using it to check data sharing in several applications which require groups, with both low performance overheads, and low annotation burden. We also discuss how these applications would have been difficult and impractical to check without groups.

2. Overview

SharC [1] is a system in which programmers add annotations, called sharing modes, to types. These modes include `private`, for objects accessible only by a single thread, `readonly` for objects that cannot be modified, and `locked(L)` (a dependent type) for objects that must only be accessed when lock L is held. SharC uses a combination of static and dynamic checking to ensure that these rules are followed. Additionally, SharC allows the programmer to specify where the sharing mode for an object changes using a *sharing cast*. SharC checks these casts using reference counting: if only one reference to an object exists, then the sharing mode of the object can be changed by atomically copying the pointer to a pointer with the new sharing mode, and nulling out the old pointer.

2.1 SharC's Limitation

With these capabilities SharC is able to describe the sharing strategy in a variety of legacy multithreaded C programs. Unfortunately, the restriction that the sharing cast may apply only to single objects prevents SharC from being easily applied to a number of common code patterns.

In this section, we will refer to a simple example in which a singly-linked list is initialized by one thread before being shared through a lock with other threads. The code for this example is given in Figure 1. This code snippet defines a type for list nodes, and gives a function that initializes a list before sharing it with other threads through a global variable protected by a lock.

The sharing strategy for this code dictates that the list is private when being constructed, and then becomes lock-protected when it is shared with other threads on line 18. We would like to simply declare that `lckL` is a pointer to a `locked() list_t`, and that its `next` field also points to `locked() list_t`'s, and that similarly `pL` is a pointer to a `private list_t` with a `next` field pointing to `private list_t`'s. However, if we do that SharC will not let us do a sharing cast from `pL` to `lckL` because that would be unsound as as there might still be a private pointer into the tail of the list now pointed to by `lckL`.

```
1  typedef struct list {
2    int data;
3    struct list *next;
4  } list_t;
5  mutex_t *Lck;
6  list_t *lckL;
7  void init() {
8    list_t *pL = NULL, *end = NULL;
9    for(int i = 0; i < 10; i++) {
10     t = malloc(sizeof(list_t));
11     t->data = i;
12     t->next = NULL;
13     if (pL == NULL) pL = t;
14     else end->next = t;
15     end = t;
16   }
17   mutex_lock(Lck);
18   lckL = pL;
19   mutex_unlock(Lck);
20 }
```

Figure 1. Code that defines a linked structure, and shows how it is initialized by one thread before being shared through a lock with other threads.

Alternately, we might try to define two different list structures, each with the appropriate annotation given explicitly on the `next` field, say `plist_t` and `lcklist_t`. Then, we could traverse the list, doing a sharing cast on each node while casting from one list node type to the other[1]. We might write something like the code in Figure 2.

```
lcklist_t locked(Lck) *convert(plist_t private *L) {
   plist_t *pt1 = L, *pt2;
   lcklist_t locked(Lck) *lckhp, **lcktp = &rot;
   while(pt1) {
     pt2 = pt1->next;
     pt1->next = NULL;
     (*lcktp) = scast (lcklist_t locked(Lck) *,pt1);
     lckhp = &((*lcktp)->next);
     pt1 = pt2;
   }
   return lckhp;
}
```

Figure 2. Code that uses SharC's single object cast to convert a `private` list to a `locked()` list. `scast` is SharC's single object sharing cast.

In this snippet `scast` is SharC's single object sharing cast. This approach has a few problems. First, we would have to compel SharC to accept the dubious cast between two different aggregate types, which it currently does not. Second, code with a similar goal may become quite cumbersome to write for anything beyond the simplest data structures. Finally, for data structures consisting of many individual objects, the above code would adversely affect run-time performance. Furthermore, condoning such an approach

[1] This tactic might require writing to `readonly` objects, if that is the intended target mode, and so will trivially violate SharC's type system, but suppose for a moment that this is not a problem.

```
1 typedef struct list {
2   int data;
3   struct list same *next;
4 } list_t;
5 mutex_t *Lck
6 list_t group(lckL) locked(Lck) * locked(Lck) lckL;
7 void init() {
8   list_t group(pL) private * private pL = NULL;
9   list_t group(pL) private * private end = NULL;
10  for(int i = 0; i < 10; i++) {
11    t = malloc(sizeof(list_t));
12    t->data = i;
13    t->next = NULL;
14    if (pL == NULL) pL = t;
15    else end->next = t;
16    end = t;
17  }
18  mutex_lock(Lck);
19  lckL = gcast (list_t group(lckL) locked(Lck) *, pL);
20  mutex_unlock(Lck);
21 }
```

Figure 3. Code that defines a linked structure, and shows how it is initialized by one thread before being shared through a lock with other threads. Bolded annotations are provided by the programmer. Unbolded annotations are inferred.

would violate SharC's design goal of requiring the programmer to write only simple type annotations and casts to specify a program's sharing strategy.

2.2 Shoal's Goals

The goal of Shoal is to allow changing the sharing mode, not only of individual objects, but also of complex data structures in one atomic action. In order to ensure the soundness of these operations, we require the programmer to add annotations that describe a *group* of objects that have the same sharing mode. A group is identified by a distinguished object called the *group leader*. Internal and external pointers to the group have distinct types such that external pointers must identify the group leader. We combine static and runtime checks to maintain the invariant that no object is referred to by pointers with different sharing modes at the same time. In brief, we ensure that all external pointers identify the same object as the group leader, and that group casts are only performed when there is a single external pointer to the group. The rest of this section explains the syntax for groups, and how our type system enforces the above semantics.

Orthogonally, to better support scientific applications, Shoal also adds a new sharing mode, `barrier` for objects protected by barrier synchronization: between any two barriers a chunk[2] of the object is either only read, or only accessed by a single thread.

2.3 Group Annotations and Casts

Groups are specified using two annotations. First, we provide a dependent type annotation, `group(p)`, which indicates that objects of that type belong to the group with leader pointed to by `p`. Second, we provide an annotation for structure field pointer types, `same`, which indicates that the structure field points into the same group as the containing structure. The `group(p)` annotation identifies *external* pointers, while the `same` annotations identifies *internal* pointers. We also provide an additional checked cast operation, `gcast`, which indicates that the sharing mode or group leader for a group of objects is changing. `gcast` ensures that there is only one external reference to a group leader, and atomically copies the value of the pointer into the new pointer with the new type, and nulls out the old pointer so that no pointers with the old type refer to the same data structure after the cast.

Consider the code snippet in Figure 3. This is our linked list example from above, but now it is fully annotated with Shoal's sharing modes, and casts. Bolded annotations must be provided by the programmer, while the unbolded annotations are inferred. First note the `same` annotation in the type of the `next` field on line 3. This indicates that `next` is an internal pointer in the same group as the enclosing structure. Next, note that `lckL` is a `locked()` pointer into a `locked()` list. The `group(lckL)` annotation indicates that `lckL` is an external pointer into the group identified by the target of `lckL`. That is, `lckL` is a pointer to the group leader. Inside of the `init()` function, `pL` and `t` are `private` pointers to `private` lists. They are both external pointers into the group identified by the target of `pL`. In the `for` loop, nodes are added to the list.

After the list is built, on line 19, the mode for the entire list is changed from `private` to `locked()`. This is safe because `pL` is the only external reference to the list, and because no other live, non-null variable refers to `pL` in its type. These checks are sufficient to ensure that there are no pointers with the wrong type. If our list building code had, e.g., stored an external reference into the middle of the list, then its type would have to refer to the group leader. If the type referred to a reference aside from `pL`, then the reference count on the group leader would be too high. If its type referred to `pL`, then the null check on pointers that refer to `pL` would fail.

2.4 The Group Leader

As mentioned, when a cast of a group is attempted, we must be able to verify that there is only one external pointer into the group. Further, for soundness reasons when a pointer mentioned in a dependent type is assigned to, pointers of types that depend on it must be null.

To enforce the first requirement, we restrict what expressions may appear as parameters to a `group()` annotation. In the annotation `group(p)`, `p` is a pointer expression built from variables and field names: the types of structure fields may refer to other fields of the same structure, and the types of locals may refer to other locals(including formal parameters) in the same function. Globals may be mentioned in the `group()` parameter, but only when they are declared `static readonly` . Finally, pointers mentioned in the `group()` parameter may not have their addresses taken. These restrictions permit us to reason locally about what values types depend on. That is, we require no alias analysis to discover what the dependencies of a type are. These are the same restrictions as those used in the Deputy [8] dependent type system for C.

The second requirement exists to ensure that the group of an object cannot change without a group cast. That is, changing the group of an object by simply changing the value of the pointer in its `group()` annotation is forbidden. To enforce this, we simply insert dynamic checks for null-ness ahead of assignments to group leader pointers on which other live pointers depend. Note that in our linked list example, on line 14 `t` is dead after the assignment, so no null-check is required for `t`. However, `pL` is live after the assignment, and so we must check that it is null beforehand. These restrictions may seem cumbersome, however they have not caused any substantial difficulties in porting our benchmarks.

In a fully typed program, every pointer type has either a `group()` annotation or a `same` annotation. However, the `group()` parameter may be null. When this is the case, the object cannot be the subject of a group cast as it clearly cannot be the group leader, however it can be extracted from the "null" group using the single-object sharing cast as described in Section 3. In our system we use the shorthand `nogroup` for `group(NULL)`. In our linked list example, the `nogroup` annotations would go on types that do not have a

[2] We break objects into 16-byte chunks, to allow concurrent non-racy updates to composite objects such as arrays.

group() or same annotation, but these have been omitted for readability. In our implementation nogroup is the default annotation. This is so that the programmer is required to make annotations only for instances of data structures where the group cast is needed.

2.5 Operations on Groups

Objects can be added to and removed from groups, while group casts can be used to change a group's sharing mode and/or merge two groups. Consider the linked list structure above. Adding a node to the list is straightforward. We simply declare a new list node pointer, and indicate that it points into the needed group as shown in the linked list example.

In our first example with list_t, we demonstrated the use of the group cast to change the sharing mode of an entire linked list. This idiom is fairly common in concurrent C programs. That is, a data structure is initialized privately to one thread before being shared among all threads either protected by a lock, or read-only. We can also merge one group into another group. This operation requires a checked cast because we do not currently support any notion of "subgroups," and so no pointers identifying the leader of the old group as their group leader may exist after the merge:

```
list_t group(Ltail) private * private Ltail;
list_t group(Lhead) private * private Lhead;
// ... nodes are added to Ltail ...
Lhead->next = gcast (list_t group(Lhead) private *,
                     Ltail);
```

Due to the few restrictions placed on internal group pointers (i.e. the same pointers), our type system does not support group splitting. However, we do support the removal of individual objects from a group using the single-object sharing cast; if the same fields of an object are null, then it can be cast to any group. The inability to split groups has not been problematic in our benchmarks.

2.6 Instrumentation

In Figure 4 we show the instrumentation that Shoal uses for its dynamic analysis. In fact, this snippet is simply meant to show logically what the instrumentation does since the actual implementation involves some amount of optimization to reduce the cost of reference counting, and some additional complexity because the reference counting must be thread safe. Since this example involves only the private and locked() modes, all of the instrumentation comes from checking that Lck is held when lckL is accessed, reference counting, and checking the reference count at the group cast. Since we can verify statically that t is dead at the assignment at line 14 and at the group cast at line 19, the only dynamic checking needed for the group() types is ensuring that pL is null before it is assigned. If t were not determined statically to be dead at the above mentioned locations, the instrumented code would contain assertions requiring t to be null.

2.7 A More Complex Example

Figure 5 gives pseudo-code for an n-body simulation using the Barnes-Hut algorithm [2]. At each simulation time-step, Barnes-Hut constructs an oct-tree based on the locations of the bodies, then computes the center of mass of each node in the oct-tree. The force on each particle is computed by walking the oct-tree; the effect of all particles within an oct-tree node can be approximated by the node's center of mass as long as this is far enough from the particle. Finally, the computed force is used to update a particle's velocity and position.

Figure 5 omits the code that builds the tree and does the calculations to highlight the data structures, sharing modes and group casts. This pseudo-code has been adapted to pthreads from a Split-C [18] program (itself inspired by the Barnes-Hut Splash2 [31] benchmark) written in an SPMD style; it uses barriers to synchronize accesses to shared data structures, and each thread has a unique id given by MYPROC. The tree is built out of node_t structures. Each node_t records the total mass of the bodies below that node and the position of their center of mass. Further, leaf nodes in the tree can point to up to eight bodies, while internal nodes have eight children. The leaves are represented in the node_t's as an index into a body_t array whose elements represent the particles. Finally, every node has a pointer to its parent node.

```
void init() {
  list_t *pL = NULL, *t = NULL, *end = NULL;
  for(int i = 0; i < 10; i++) {
    decrc(t);
    t = malloc(sizeof(list_t));
    incrc(t);
    t->data = i;
    t->next = NULL;
    if (pL == NULL) {
      assert (pL == NULL);
      decrc(pL);
      pL = t;
      incrc(pL);
    }
    else {
      decrc(end->next);
      end->next = t;
      incrc(end->next);
    }
    decrc(end);
    end = t;
    incrc(end);
  }
  mutex_lock(Lck);
  sharc_lock_acquire(Lck);
  assert(sharc_has_lock(Lck));
  // The following six lines are performed atomically
  assert(refcount(pL) == 1);
  decrc(lckL);
  lckL = pL;
  incrc(lckL);
  decrc(pL);
  pL = NULL;
  sharc_lock_release(Lck);
  mutex_unlock(Lck);
}
```

Figure 4. Our linked list example shown after instrumentation by Shoal. Since the example is using only the private and locked() modes, the only instrumentation is for reference counting, checking that locks are held, and checking the group cast.

The function run(), executed by each thread, shows an outline of how the simulation runs. First, the initial positions and velocities of the bodies are written into the array of body_t structures(line 26) by one of the threads. This initialization occurs privately to this thread. This is reflected by the private sharing mode of the bodies array on line 22.

Next, the threads enter the main loop. First, an oct-tree is built by the same thread that initialized the bodies(line 30). While the tree is being built, the parent pointers in the nodes are filled in. In the last step of BuildTree(), the tree is walked up from each leaf node to calculate the mass and center of mass for each node[3]. During this process the parent nodes are nulled out, as they will not be used again.

After the tree is built, it will no longer need to be written. Further it will need to be read by the other threads. This sharing

[3] We note here that it is sometimes necessary when using groups to pass an unused pointer to the group leader to recursive functions. This has not caused problems in our benchmarks.

```
 1 typedef struct node {
 2   double mass;
 3   double pos[3];
 4   struct node same *parent;
 5   int bodies[8];
 6   struct node same *children[8];
 7 } node_t;
 8
 9 typedef struct body {
10   double mass;
11   double pos[3];
12   double vel[3];
13   double acc[3];
14 } body_t;
15
16 void BuildTree(node_t group(root) private *root,
17                body_t private *Bodies);
18
19 void run() {
20   node_t group(root) private *root;
21   node_t group(roRoot) readonly *roRoot;
22   body_t private *bodies;
23   body_t barrier *barBodies;
24
25   if (MYPROC == 0)
26     initBodies(bodies);
27   while (not_done) {
28     if (MYPROC == 0) {
29       root = malloc(sizeof(node_t));
30       BuildTree(root,bodies);
31       roRoot =
32         gcast(node_t group(roRoot) readonly *,root);
33       barBodies =
34         scast(body_t barrier *,bodies);
35     }
36     barrier();
37     // Share roRoot and barBodies, then calculate
38     // new body positions.
39     barrier();
40     if (MYPROC == 0) {
41       root =
42         gcast(node_t group(root) private *,roRoot);
43       bodies =
44         scast(body_t private *,barBodies);
45       free_tree(root);
46     }
47   }
48 }
```

Figure 5. An oct-tree is built privately to one thread, and then the entire tree is cast to `readonly` to be shared with all other threads.

mode change is implicit in the original program through the use of the `barrier()` call, but with Shoal such changes must be explicit. Hence, we use a group cast to make the entire tree `readonly` (line 32). This cast succeeds because there is only one external reference to the whole oct-tree, and because no other pointers aside from `root` mention `root` in their types. Further, no other pointers aside from `roRoot` mention `roRoot` in their types, so it is also safe to write `roRoot`. If `BuildTree` had saved an external reference into the tree, then this external reference would have been dependent on some pointer to the group leader. This unsafe condition would then be caught during the checked cast because the reference count to the group leader would be greater than one.

We also cast the `bodies` array(line 34). We use the `barrier` sharing mode because different threads "own" different parts of the array. The thread that owns a body calculates its new position, velocity, and acceleration based on the masses of nearby bodies (distant bodies are not accessed as their effects are summarised

Program	P	::=	$\Delta \mid P; P$
Definition	Δ	::=	$x : \tau \mid t = (f_1 : \phi_1, \ldots, f_n : \phi_n)$
		$\mid$	$f()\{x_1 : \tau_1, \ldots, x_n : \tau_n; s\}$
Dependent Type	τ	::=	$m\langle\ell\rangle\ t$
Field Type	ϕ	::=	$t \mid \tau$
Sharing Mode	m	::=	`locked` | `private`
Statement	s	::=	$s_1; s_2 \mid \text{spawn } f() \mid \text{lock } x \mid \text{unlock } x$
		$\mid$	$\ell := e\ [\text{ when } \omega_1, \ldots, \omega_n\ ]$
		$\mid$	**wait** | **done**
L-expression	ℓ	::=	$x \mid x.f$
Expression	e	::=	$\ell \mid \text{new} \mid \text{gcast } x$
Predicate	ω	::=	$\mathit{oneref}(x) \mid m(x) \mid \ell_1 = \ell_2 \mid \ell = \text{null}$
Identifiers			f, x, t

Figure 6. A simple imperative language. Elements in bold are only used in the operational semantics.

in the oct-tree). Some parts of the body structure are `readonly`, and other parts are `private`. This fact cannot be represented by our static type system, but we can check the desired access pattern using the `barrier` sharing mode. The cast of bodies can use the single-object sharing cast (`scast`) as the `body_t` type contains no `same` pointers.

Following these casts, the pointers to the body array and the tree root are shared through globals with the other threads, and the simulation is updated. Subsequently, these pointers are nulled out in all threads but one, and more casts are used to indicate that the body array and oct-tree are private again (lines 41 and 43). The oct-tree can then be destroyed (line 45).

If this program compiles and runs without errors with Shoal, then we know that no pointers exist to the oct-tree or to the bodies with the wrong type, that there are no data races on the bodies while they are protected by the barrier synchronization (or on other objects in other sharing modes), and that only one thread accesses the oct-tree or the bodies while they are in the `private` mode.

On this example, Shoal's dynamic checking incurs a runtime overhead of 38% when running a simulation of 100 time steps for approximately 16000 bodies. Most of this overhead is due to the cost of reference counting.

2.8 Static vs. Dynamic Checking

Shoal does very little dynamic checking beyond what is already done by SharC, i.e. reference counting, and checking for the `locked` sharing modes. As mentioned above, our `group()` type is dependent, and as one might expect, some of the type checking must be done at runtime. In our case, this entails inserting a small number of checks to make sure that certain pointers are null. In practice the number of checks is small, and our timing primitives are not precise enough to measure their affect on the performance of our benchmarks.

Shoal's `barrier` mode is checked at runtime. For objects in the `barrier` mode, in between barriers, dynamic checks are inserted that require these objects to be either private or read-only between barriers. At each barrier, the analysis is reset to reflect the fact that the ownership of `barrier` mode objects changes at barriers.

All other components of our analysis are done statically.

3. The Formal Model

Shoal's formalism is derived from that SharC [1], with significant changes to handle data structures, and our notion of groups and group casts. Our language (Figure 6) consists of global variable ($x : \tau$), C-like structure ($t = \ldots$) and function ($f()\{\ldots\}$) definitions (we assume all identifiers are distinct). Our global variables are

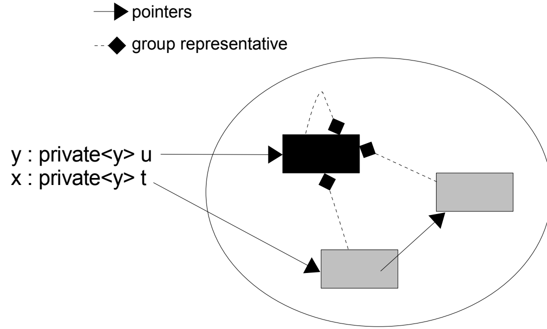

Figure 7. A simple group

unusual: for each global variable $x : \tau$ a structure instance o_x of type τ is allocated at program startup, and newly created threads have a *local* variable $x : \tau$ that is initialized to o_x. This allows sharing between our threads while simplifying our semantics by allowing all variables to be treated as local.

Figure 7 shows typical type declarations for two variables `x` and `y`: a type τ of the form *m*<*x*> *t* represents a pointer to the structure named *t*, that can be null. The sharing mode *m* is the sharing mode of the pointer's target; our formalization supports `private` (accessible to only one thread) and `locked` (each structure protected by an implicit lock, Java-style). As we described earlier, each group is identified by a distinguished *leader* object. In our type system, all pointer types must include the name *x* of a variable or field that points to the leader (the syntax allows for arbitrary lvalues, but these can only appear during type-checking and in the operational semantics). For instance, in Figure 7 `y` points to the leader for `x`'s target. In the rest of this section, we simply call `y` `x`*'s leader*. Note that the leader of `y` is `y` itself (this is required for soundness), however it is legal to have multiple pointers to the group leader. For instance, we could add declarations

```
y1 : private<y1> u
x1 : private<y1> t
```

and legally assign `y` to `y1` and `x` to `x1`. Thus, the declaration (`y : m<x> t`) is equivalent to the C language declaration (`t m group(x) * private y`) since all variables in the formalism are locals. Our pointer types depend on variables and are thus a form of dependent types. To ensure soundness in the presence of mutation, we use similar type rules to the Deputy system [8] (see below).

Structures consist of a set of typed fields $f : \phi$, where ϕ is either a dependent type τ or an unqualified pointer to a structure *t*. In the first case, the type τ can only depend on other fields (as in Deputy) and cannot be `private`[4], in the second case the pointer is a "same group" pointer, i.e. it's target is in the same group and has the same sharing mode as the structure itself. For example

```
list = (data: locked<data> stuff, next: list)
x : private<x> list
```

declares a linked-list type where the list node structures are in a single group. Thus, these declarations are equivalent to the C language declarations:

```
struct list {
  stuff locked(data) group(data) *data;
  struct list same *next;
}
```

[4] This requirement can be relaxed to forbidding non-private pointers to such structures.

```
struct list private group(x) * x;
```

The variable `x` is thus a private list with locked contents: `x`, `x.next`, `x.next.next`, ... are all private objects in the same group, while the list contents `x.data`, `x.next.data`, ... are in separate groups protected by locks.

Functions *f* consist of local variable definitions and a sequence of thread-spawning, locking and assignment statements. Assignments are guarded by runtime checks (ω) which check sharing modes are respected ($m(x)$), compare lvalues ($\ell = \ell'$ and $\ell =$ null) and assert that *x* is the sole reference to a particular object (*oneref*(x)). These runtime checks are added automatically during type checking. The most important expression is gcast *x* which performs a *group cast* on *x*'s target. For instance:

```
y : locked<y> list
y := gcast x
```

casts our list `x` whose node structures were `private` into a list `y` whose node structures are protected by locks. A group cast can also merge two groups:

```
y : locked<z> list
y = gcast x
z.next = y
```

Our formalism can readily be extended with primitive types, further sharing modes and an scast *x* expression to extract a single object from a group (under the condition that all same-group fields are null). However, our formalism does not currently support splitting a group into two arbitrary parts.

3.1 Static Semantics

Figure 8 presents the type-checking rules for our language. These rules both check a program for validity and rewrite it to include necessary runtime checks. Also, we restrict assignments to the forms $x := e$ and $x.f := y$. To simplify exposition, we assume in the rules that $\Gamma_G(x)$ gives the type of global variable *x*, *T* is the set of all structure types, and $t(f)$ gives the type of field *f* of structure *t*.

The GLOBAL, STRUCTDEF and THREAD rules enforce the restriction that variable types are only dependent on other variable types, while field types are only dependent on other fields of the same structure. Furthermore, globals cannot be `private` and structures cannot contain explicitly `private` fields, as having a non-private pointer to a structure with a private field would be unsound. The DEPENDENT rule requires that the leader of a group uses itself as leader.

Reading or writing lvalue ℓ (READ, WRITE) requires three sets of runtime checks. First, if $\ell = x.f$ we must enforce *x*'s sharing mode. Second, we must ensure that the lvalues denoting the assignment's source and target group match after the assignment. Third, the function D adds runtime checks to ensure that any variables or fields dependent on the assignment's target ℓ' are null before the assignment, as the assignment would otherwise be unsound. The scoping rules enforced in GLOBAL, STRUCTDEF and THREAD make these checks tractable: if $\ell' = x.f$, fields dependent on *f* must be in the same structure instance, while variables dependent on $\ell' = x$ must be variables of the same function. Finally, these checks remain sound even in the presence of concurrent updates by other threads: variables cannot be modified by other threads, while the runtime check ω that enforces $\ell = x.f$'s sharing mode also guarantees that any other fields of *x* can be accessed without races.

A group cast of *y* can be performed if *y* is the only reference to the target object. Because the group cast nulls-out *y*, the dependent-type restrictions (D) must hold for *y*, as in a regular assignment to *y*.

$$
\begin{array}{rcl}
M, id : \text{lock } x & \xrightarrow{s} & M[M_v(id.x) \xrightarrow{L} id] \textbf{ if } M_L(M_v(id.x)) = 0 \\
M, id : \text{unlock } x & \xrightarrow{s} & M[M_v(id.x) \xrightarrow{L} 0] \textbf{ if } M_L(M_v(id.x)) = id \\
M, id : x := \text{new} & \xrightarrow{s} & extend(M[id.x \xrightarrow{v} a], id, a, rtype(M[id.x \xrightarrow{v} a], id, M_\rho(id.x))) \textbf{ where } a = \max(\text{dom}(M)) + 1 \\
M, id : x := \text{gcast } y & \xrightarrow{s} & gcast(M', id, c, rtype(M', id, M_\rho(id.x))) \textbf{ where } c = M_v(id.y), M' = M[id.x \xrightarrow{v} c, id.y \xrightarrow{v} 0] \\
M, id : \ell := \ell' & \xrightarrow{s} & M[lval(\ell) \xrightarrow{v} M_v(lval(\ell'))]
\end{array}
$$

$$
\frac{M_v(lval(\ell)) = M_v(lval(\ell'))}{M, id \models \ell = \ell'} \qquad \frac{M_v(lval(\ell)) = 0}{M, id \models \ell = \text{null}} \qquad \frac{}{M, id \models \texttt{private}(x)} \qquad \frac{M_L(M_v(id.x)) = id}{M, id \models \texttt{locked}(x)} \qquad \frac{|\{b.f | M_v(b.f) = M_v(id.x)\}| = 1}{M, id \models oneref(x)}
$$

(SIMPLE STATEMENT)

$$
\frac{M, id : s_1 \xrightarrow{s} M'}{M, \{(id, s_1; s_2)\} \oplus S \rightarrow M', \{(id, s_2)\} \cup S}
$$

(RUNTIME CHECK)

$$
\frac{M, id \models \omega_1}{M, \{(id, \ell := e \text{ when } \omega_1, \omega_2 \ldots, \omega_n; s)\} \oplus S \rightarrow M, \{(id, \ell := e \text{ when } \omega_2 \ldots, \omega_n; s)\} \cup S}
$$

(THREAD CREATION)

$$
\frac{id' = \max(\text{dom}(M)) + 1 \qquad M' = extend(M, id', id', M_\rho(id_f)) \qquad M'' = M'[id' \xrightarrow{v} M_v(id_f)]}{M, \{(id, \text{spawn } f(); s), (id_f, \textbf{wait}; s_f)\} \oplus S \rightarrow M'', \{(id', s_f; \textbf{done}), (id, s), (id_f, \textbf{wait}; s_f)\} \cup S}
$$

(THREAD DESTRUCTION)

$$
\frac{M' = M \backslash id}{M, \{(id, \textbf{done})\} \oplus S \rightarrow M', S}
$$

$$
lval(M, id, x) = id.x \quad lval(M, id, x.f) = M_v(id.x).f \textbf{ if } M_v(id.x) \neq 0
$$
$$
rtype(M, id, m\text{<}x\text{>}\ t) = m\text{<}M_v(id.x)\text{>}\ t
$$
$$
extend(M, id, a, \rho) = M[a \rightarrow (\rho, id, 0, \lambda f.0)]
$$
$$
gcast(M, id, c, \rho) = M' \textbf{ where } \begin{cases} M'_\rho(a) = \rho \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_\rho(a) \textbf{ otherwise} \\ M'_o(a) = id \textbf{ if } M_\rho(a) = m'\text{<}c\text{>}\ t \wedge c \neq 0, M_o(a) \textbf{ otherwise} \\ M'_v(a) = M_v(a), M'_L(a) = M_L(a) \end{cases}
$$

Figure 9. Operational semantics.

3.2 Operational Semantics

The parallel operational semantics of Figure 9 models the fine-grained interleaving of threads in a shared memory setting. The shared memory $M : \mathbb{N}^+ \rightarrow \rho \times \mathbb{N} \times \mathbb{N} \times (name \rightarrow \mathbb{N})$ maps a cell's address to its runtime type, owner, locker, and value. The runtime types ρ are of the form $m\text{<}a\text{>}\ t$, where a is the address of the cell's group leader. The owner is the thread identifier (an integer) that owns the cell if it's sharing mode is `private`. The locker is the thread identifier that currently holds a lock on the cell, or 0 if the cell is unlocked. Finally, the value is a map from field names to cell addresses. We use the notation $M_\rho(a)$, $M_o(a)$, $M_L(a)$ and $M_v(a)$ to denote respectively the type, owner, locker and value of cell a, while $M[a \xrightarrow{\rho} \rho]$, $M[a \xrightarrow{o} n]$, $M[a \xrightarrow{L} n]$ and $M[a \xrightarrow{v} v]$ represent the corresponding updates to cell a. For convenience, we also write $M_v(a.f)$ and $M_\rho(a.f)$ to return respectively the value and type of field f of cell a, and $M[a.f \xrightarrow{v} b]$ to update field f of cell a. Note that $M_\rho(a.f)$ returns the static type τ of field f, not the runtime type of cell $M_v(a.f)$.

Thread identifiers are simply the address of a memory cell whose value is a map of the thread's variable names to their values. States of the operational semantics are of the form

$$M, (id_1, s_1), \ldots, (id_n, s_n)$$

representing a computation whose memory is M and which has n threads, each identified by address id_i and about to execute s_i. To handle thread creation, each thread definition f is represented by a *prototype thread* of the form $(id_f, \textbf{wait}; s_f)$ where s_f is the body of thread f and $M(id_f)$ is a memory cell with the initial state of f's variables: $M_v(id_f.x)$ is null for all local variables of f and a pointer to the preallocated structure instance o_x for global variable x.

Transitions between states are defined with a number of helper judgments and functions:

- $M, id : s \xrightarrow{s} M'$: execution of simple statement s by thread id updates the memory M to M'.
- $M, id \models \omega$ holds if runtime check ω is valid.
- $lval(M, id, \ell)$ converts a static lvalue ℓ of thread id to a runtime lvalue of the form $a.f$ (field f of cell a), and is undefined if ℓ requires dereferencing a null pointer.
- $rtype(M, id, \tau)$ converts a static type τ of thread id to the corresponding runtime type ρ.
- $extend(M, id, \rho)$ represents allocation of a cell of runtime type ρ by thread id.
- $gcast(M, id, c, \rho)$ performs a group cast to ρ in thread id of the group whose leader is $M(c)$.

The rules are mostly straightforward, but a few points are worth noting. We use $\oplus$ to denote disjoint union ($A = B \oplus C \Rightarrow B \cap C = \emptyset$). Individual runtime checks are executed atomically, but involve at most one lvalue of the form $x.f$. Other threads may execute between runtime checks and between the checks and the assignment they protect. The $oneref(x)$ check is computed by heap inspection, but we implement it in practice using reference-counting. We could exclude non-dependent fields from $oneref(x)$ check, this would correspond to not reference-counting **same** pointers.

A group cast of a variable y whose value is null should have no effect. To ensure this, *gcast* never updates cells whose type is $m\text{<}0\text{>}\ t$. Effectively, group 0 corresponds to our `nogroup` C-level annotation: once in group 0, you can no longer be subject to a group cast. Our formalism does allows casts to group 0, and allocations in group 0. Finally, and most importantly, the type and owner fields of M are not required in an actual implementation, thus *gcast* has no runtime cost.

$\vdash P \Rightarrow P'$	P' is identical to P except for runtime checks that ensure P's sound execution.

(DEFINITIONS)
$$\frac{\vdash \Delta_1 \Rightarrow \Delta_1' \qquad \vdash \Delta_2 \Rightarrow \Delta_2'}{\vdash \Delta_1; \Delta_2 \Rightarrow \Delta_1'; \Delta_2'}$$

(GLOBAL)
$$\frac{\Gamma_G \vdash m\texttt{<}y\texttt{>}\ t \qquad m \neq \texttt{private}}{\vdash x : m\texttt{<}y\texttt{>}\ t \Rightarrow x : m\texttt{<}y\texttt{>}\ t}$$

(STRUCTDEF)
$$\frac{[f_1 : \phi_1, \ldots, f_n : \phi_n] \vdash \phi_i \qquad \phi_i \neq \texttt{private<}f\texttt{>}\ u}{\vdash t : (f_1 : \phi_1, \ldots, f_n : \phi_n) \Rightarrow t : (f_1 : \phi_1, \ldots, f_n : \phi_n)}$$

(THREAD)
$$\frac{\Gamma = \Gamma_G[x_1 : \tau_1, \ldots, x_n : \tau_n] \qquad \Gamma \vdash \tau_i \qquad \Gamma \vdash s \Rightarrow s'}{\vdash f : \{x_1 : \tau_1, \ldots, x_n : \tau_n; s\} \Rightarrow f : \{x_1 : \tau_1, \ldots, x_n : \tau_n; s'\}}$$

$\Gamma \vdash \phi$	Type ϕ is valid in environment Γ.

(SAME)
$$\frac{t \in T}{\Gamma \vdash t}$$

(DEPENDENT)
$$\frac{\Gamma(x) = m\texttt{<}x\texttt{>}\ u \qquad t \in T}{\Gamma \vdash m\texttt{<}x\texttt{>}\ t}$$

$\Gamma \vdash \ell : \tau, \omega$	In environment Γ, ℓ is a well-typed lvalue with type τ assuming ω holds.

(NAME)
$$\frac{\Gamma(x) = m\texttt{<}y\texttt{>}\ t}{\Gamma \vdash x : m\texttt{<}y\texttt{>}\ t, \epsilon}$$

(SAME FIELD)
$$\frac{\Gamma(x) = m\texttt{<}y\texttt{>}\ t \qquad t(f) = t'}{\Gamma \vdash x.f : m\texttt{<}y\texttt{>}\ t', m(x)}$$

(DEPENDENT FIELD)
$$\frac{\Gamma(x) = m\texttt{<}x\texttt{>}\ t \qquad t(f) = m'\texttt{<}g\texttt{>}\ t'}{\Gamma \vdash x.f : m'\texttt{<}x.g\texttt{>}\ t', m(x)}$$

$\Gamma \vdash s \Rightarrow s'$	In environment Γ statement s compiles to s', which is identical to s except for added runtime checks.

(NEW)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x := \text{new} \Rightarrow x := \text{new when } \mathrm{D}(\Gamma, x)}$$

(GROUP CAST)
$$\frac{x \neq y \qquad \Gamma(x) = \tau \qquad \Gamma(y) = m'\texttt{<}y\texttt{>}\ t}{\Gamma \vdash x := \text{gcast } y \Rightarrow x := \text{gcast } y \text{ when } oneref(x), \mathrm{D}(\Gamma, x), \mathrm{D}(\Gamma, y)}$$

(READ)
$$\frac{\Gamma(x) = m\texttt{<}x'\texttt{>}\ t \qquad \Gamma \vdash \ell : m\texttt{<}\ell'\texttt{>}\ t, \omega}{\Gamma \vdash x := \ell \Rightarrow x := \ell \text{ when } \omega, x'[\ell/x] = \ell', \mathrm{D}(\Gamma, x)}$$

(WRITE)
$$\frac{\Gamma \vdash x.f : m\texttt{<}\ell\texttt{>}\ t, \omega \qquad \Gamma(y) = m\texttt{<}y'\texttt{>}\ t}{\Gamma \vdash x.f := y \Rightarrow x.f := y \text{ when } \omega, \ell[y/x.f] = y', \mathrm{D}(\Gamma, x.f)}$$

Runtime checks for dependent-type assignments

$$\mathrm{D}(\Gamma, x) = \{y = \text{null} \mid x \neq y \wedge \Gamma(y) = m\texttt{<}x\texttt{>}\ t\}$$
$$\mathrm{D}(\Gamma, x.f) = \{x.g = \text{null} \mid \Gamma(x) = m\texttt{<}y\texttt{>}\ t \wedge f \neq g \wedge t(g) = m\texttt{<}f\texttt{>}\ t\}$$

Figure 8. Typing judgments. We omit the straightforward rules for non-assignment statements.

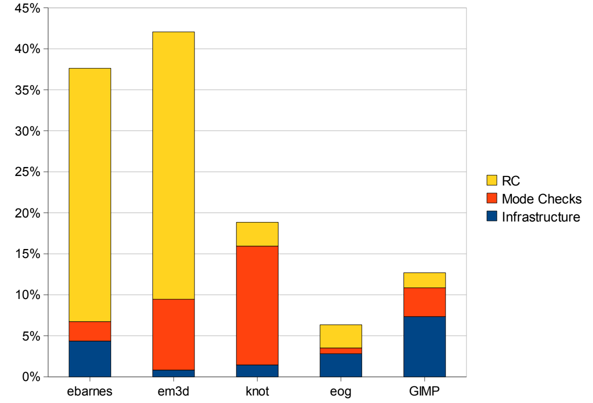

Figure 10. The breakdown of overheads in Shoal. For each benchmark, we show the cost of reference counting(RC), the dynamic checks for sharing modes (Mode Checks), and the cost of using our compiler front-end and infrastructure.

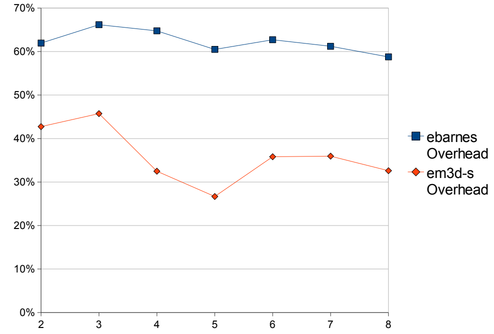

Figure 11. The overhead incurred by Shoal as the number of threads increases and the workload remains fixed for the ebarnes and em3d-s benchmarks.

3.3 Soundness

We have proved the following theorems, that show that type safety is preserved and that the program respects the behavior specified by the sharing mode declarations. The proofs appears in full in the appendix.

THEOREM 1. *Soundness.* Let P be a program with $\vdash P \Rightarrow P'$ and M_0, S_0 be the initial memory and threads for program P'. Let $M, \{(id_1, s_1), \ldots, (id_n, s_n)\}$ be a state reachable in i steps from M_0, S_0. Then types and owners are consistent in M, and all statements s_i typecheck.

THEOREM 2. *Safety of `private` accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{private<}b\texttt{>}\ t$ then $M_o(a) = id$.

THEOREM 3. *Safety of `locked` accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{locked<}b\texttt{>}\ t$ then $M_L(a) = id$.

4. Evaluation

We applied Shoal to 5 interesting applications, the largest of which approaches 1 million lines of code. All but one of these benchmarks

Benchmark						Performance	
Name	Conc.	Lines	Anns.	Casts	Max Group	Orig.	Shoal
ebarnes	2	3k	60	16	16384	6.26s	38%
em3d-s	2	1k	42	2	100000	3.59s	42%
knot	3	5k	60	17	75	0.69s	19%
eog	4	38k	67	38	20	1.42s	6%
GIMP	4	936k	37	32	8	9.67s	13%

Table 1. Benchmarks for Shoal. For each test we show the maximum number of threads running concurrently(Conc.), the size of the benchmark including comments (Lines), the number of annotations we added (Anns.), sharing casts required (Casts), and the maximum size of a group in a run of the benchmark. We also report the time overhead caused by Shoal

use multithreading to improve performance; we have included in our benchmarks a webserver that uses a thread pool to serve clients in parallel and hide I/O latency.

None of the benchmarks required more than a few days to make sufficient annotations to suppress a small number of false error reports and achieve good performance. The amount of human program understanding required to make these annotations is not especially burdensome.

The goal of these experiments is to show that Shoal allows practical checking of data sharing in a substantially wider range of programs than it would be able to without groups, while maintaining low overhead. In the course of running our benchmarks, we found no bugs, and the only sharing errors found were benign data races on flags used for condition variables. These races would at worst cause a thread to needlessly call a `condition_wait()` function an extra time. For these flags we used Shoal's `racy` annotation to suppress the false error reports. It is not surprising that we did not find more serious bugs because our testing of the benchmark applications was intended only to measure the performance overhead seen in typical use-cases.

4.1 Implementation

We extended the static type system of SharC to include our `group()` and `same` annotations. Further, we added our group cast operation, and extended the dynamic analysis with the null checks required by our dependent group types. We also implemented the `barrier` mode by generalizing SharC's existing dynamic analysis.

We reduced the additional annotation burden presented by the addition of groups by automatically applying common default annotations to unannotated objects. First, the default group annotation on pointer types that need one is `nogroup`. This way, the programmer must only annotate objects needing a group annotation if they may eventually be involved in a group cast. Secondly, the default annotation on structure fields in objects that may be subject to a group cast is `same`. This prevents SharC from forbidding casts of these structures.

4.2 Benchmarks

The results of our experiments are reported in Table 1. The figures we report are averages over 50 runs of each benchmark. These experiments were performed on a machine with a dual core 2GHz Intel® Xeon® 5130 processor with 2GB of memory. Excepting the scientific benchmarks, the runtime overhead imposed by Shoal is less than 20%. The higher overhead in the scientific benchmarks is due to reference count updates in the inner loops. We also observed a reasonable annotation burden, ranging from one annotation or cast per 25 lines in the small benchmarks to less than one per 10 000 lines on the largest.

Ebarnes was presented in Section 2.7. Em3d-s is another adapted Split-C scientific program that models the propagation of electromagnetic waves through objects in three dimensions. In this simulation the system is modeled as a bipartite graph. One partition models the electric field, and the other partition models the magnetic field. At the beginning of the simulation the graph is constructed privately to one thread. Then, a sharing cast is used to indicate that the graph is shared with other threads with accesses synchronized by barriers. To allow the sharing mode of the entire graph to change, the nodes and edges form a group whose leader is the structure containing pointers to the lists of nodes and edges.

Knot [29] is a simple multithreaded webserver. It maintains a thread pool for processing client requests. The threads share an in-memory cache of the files on disk. The cache is implemented as a hash table and is protected by a lock. Each call to the hash table API is made while the lock is held. Therefore, we cast the entire hash table to `private` before passing it to the hash table functions. The primary advantage of this approach is that it allows the hash table API to be used both in single-threaded and multi-threaded contexts. The benchmark for knot involves two clients simultaneously making 10k requests for small files that fit entirely in the in-memory cache. The webserver and clients both ran on the same host. The goal of this setup is to minimize the extent to which the benchmark is I/O bound. The overhead for knot in Table 1 is reported as the percent increase in CPU utilization. However, runtime for the benchmark did not increase significantly, nor did the throughput decrease significantly. Because the sharing mode of the entire in-memory cache changes from locked to private and back, the hash table data structure used for the cache forms a single group along with all cache entries. The hash table object containing pointers to the array used for the hash table and other metadata is the group leader.

Eog is the Eye-of-Gnome image manipulation program distributed with the Gnome desktop environment. It can open, display, and rotate many different image formats. Eog uses threads to hide the latency of the operations performed on images from the user interface. These operations include loading and saving images, transforming them, and creating image thumbnails. The GUI places "Job" data structures in a lock protected queue. When worker threads take jobs off of the queue, they must cast the entire Job data structure, which contains the image, and image metadata, to the `private` mode. Each Job structure and the objects to which it has pointers are placed in a group with the Job structure itself being the group leader. For this benchmark we measured the CPU time needed to open an image, rotate it, and then save it back to disk.

The GIMP is also an image manipulation program, but with many more features than Eye-of-Gnome. In particular, it is scriptable, and capable of many more sophisticated image transformations. On loading an image, the GIMP divides it up into distinct sections, which are stored in data structures called "Tiles." To perform a transformation, the GIMP creates a thread pool and an iterator over tiles that is protected by a lock. When a tile is in the iterator it is in the locked mode, but when a thread removes it from

the iterator it casts it to the `private` mode. The Tile object contains pointers that would be cumbersome to cast individually, so each Tile object and the objects to which it has pointers are made a group with the Tile object itself being the group leader. For this benchmark, we wrote a script that opened an image, performed 20 separate "Blur" operations, and then saved the image back to disk.

4.3 Scaling

In order to see how Shoal scales with the number of threads, we ran the two scientific benchmarks on a server machine with 4GB of memory and two quad-core Intel® Xeon® E5462 processors running at 2.8GHz. As shown in Figure 11, we observed that the overhead incurred by our system did not increase significantly as the number of threads ranged from two to eight while the workload was held constant. The difference in relative overhead on this machine vs the dual core machine used in the main experiments is due to differential speedup in the application and overhead: absolute overhead for both ebarnes and em3d-s decreased by 30%, as expected from the clock speed increase, while the application code sped up by 51%–59%, much more than the clock speed increase. We suspect, though have not confirmed, that the application code gets an extra advantage from the larger L2 cache (2x6MB vs 4MB) and memory bus speed (1600MHz vs 1333MHz).

The main scaling bottleneck in Shoal is that only one thread may be computing a reference count at any time. This could impact programs performing many group or sharing casts. However, this effect was not observed in our experiments.

4.4 Sources of Overhead

Figure 10 shows the sources of performance overhead for our benchmarks. They are broken down into concurrent reference counting, the Shoal runtime checks, and various infrastructure costs. Infrastructure costs are incurred by the use of the CIL [21] compiler front end[5], a custom malloc implementation needed for concurrent reference counting, and the need to null out pointers on allocation for soundness reasons.

We observe substantial reference counting overhead in the ebarnes and em3d-s benchmarks due to the reference count updates needed when building and traversing the oct-trees and graph respectively. In eog and the GIMP, the overall low overhead makes the cost of our custom rumtime more apparent.

4.5 The Need for Groups

In Table 1, we also report the maximum size of a group in each of our benchmarks. We interpret group size to be a rough measure of how costly it would be to port a program to SharC without groups, both in terms of performance and programmer effort. This interpretation is justified in the following ways. Without groups, it would be necessary to cast each object in a group individually. In order to meet the single reference requirement for the sharing cast, it is necessary to completely remove an object from a data structure before casting it. If a data structure has cycles, the code to perform this operation can be complex, and goes far beyond the low level of effort needed to annotate a program. Further, the additional data structure traversal, reconstruction, and reference counts would cause additional programmer and performance overhead.

5. Related Work

The most closely related work is our paper describing the original SharC system [1]. This paper described our extensions to SharC's type system with support for handling complex data structures. Many other researchers have investigated both type systems for describing aggregates of objects, and methods for making concurrent programs safe.

5.1 Ownership and Region Type Systems

Ownership types have been used to statically enforce properties such as object encapsulation, race-freedom, and memory safety [3]. An ownership type system statically guarantees that every object has a unique owner and that the ownership relation is a forest of trees. The type system allows multiple pointers to an object but statically checks that only the pointer from its owner has special privileges. For instance, an ownership type system for object encapsulation prevents unrestricted accesses to an object by checking that all accesses happen via the pointer from its owner [5]. Likewise, an ownership type system for race-freedom requires each access to an object *o* to be protected by a lock held on its *root* owner, which is the object at the root of the tree in the ownership relation forest that contains *o* [4]. Finally, an ownership type system can be combined with a region type system and used for region-based memory management [6] in which each object is allocated in the same region as its owner. The resulting type system is more restrictive and enforces memory safety by forbidding pointers from outside the region to objects inside the region which enables it to safely free all objects in the region whenever the region handle is freed. Further, dynamic analyses have been developed to help programmers visualize ownership relationships in their programs [23]. In these systems, instead of requiring programmers to annotate instances of encapsulation, which is the common case, the programmer is shown the cases in which encapsulation is violated, which is typically less common.

The relationship of a group leader to objects in its group is analogous to that of an owner to objects it owns. Groups however are more general in that objects can be added to or removed from them and group leaders can change dynamically, in contrast to the ownership relation, which cannot be changed dynamically. We have found that this ability is crucial for applying an ownership-like system to large C programs.

Region type systems have strong similarities to groups: they use types to identify a collection of objects, and allow global operations (typically deallocation) to be performed on that group. RC [15] is the closest to our groups: it uses reference counting to track pointers into regions, and has a `sameregion` qualifier to identify pointers that stay within a region. However, as with other regions systems, region membership is fixed at allocation time and tracked explicitly at runtime, regions cannot be merged and `sameregion` is mostly a performance hint to avoid the need for reference-counting. Tofte and Talpin's statically checked region type system [28] places further restrictions on region lifetimes and pointers between regions: essentially, region lifetimes are nested and pointers are only allowed from longer-lived regions to shorter-lived regions, but not vice-versa. Crary and Walker's [9] use of capabilities relaxes these restrictions somewhat, but still does not allow unrestricted pointers between regions. Effectively, Shoal's use of reference-counting allows a runtime-check to recover the linearity property of a group (region) reference that the static type systems must ensure at compile-time.

The authors of Cyclone [17] note the similarity in their type system between types for region membership and types for lock protection. Indeed, they go so far as providing a shorthand for when objects in the same region are protected by the same lock. Shoal expands on this idea by allowing objects belonging to the same data structure (which may be unrelated with respect to memory allocation concerns) to be related not only by locking, but also a variety of mechanisms for safe concurrency. Further, we note

[5] Using the CIL front end can cause changes in performance. CIL performs some transformations (e.g. introducing additional temporary variables) that can both enable and prevent certain C compiler optimizations.

that Shoal would likely benefit from some of the polymorphism in Cyclone's type system, but we leave this extension for future work.

5.2 Safe Concurrency Checking

There has been much work on finding concurrency related bugs in multi-threaded programs. Researchers have attempted to find and prevent data races, and atomicity violations, using both static [20, 24, 14, 22, 30, 12, 17, 19] and dynamic [25, 32, 11, 13, 7] analysis. SharC differs from these approaches because it attempts to find violations of a programmer specified sharing strategy rather than searching for violations using lock-set or happens-before based heuristics. The original SharC paper discusses in detail its relation to these traditional concurrency checking methods.

We also wish to mention a few other related projects. First, race freedom can be checked by translation to a linear program based on fractional capabilities [27]. Since linear programming instances can be solved in parallel, this technique may be able to scale to large programs. However, the possibly substantial collection of warnings may be difficult to investigate because analysis results do not indicate what locks may be held at each program point. Shoal scales to large programs, and gives detailed error reports.

Some other tools are also guided by programmer annotations. LockLint [10] is a static system that checks that locks are used consistently to protect program variables. It uses annotations to suppress false warnings, to describe hierarchical locking, and procedure side-effects, and as documentation. Further, the Fluid Project [16] has investigated a system for Java that allows a programmer to make annotations describing hierarchical regions, aliasing intent, and a locking strategy that can apply to regions, among other annotations. Shoal differs from these systems primarily because it allows the sharing mode for data structures to change through sharing and group casts as the program executes.

6. Conclusion

In this paper, we have presented groups, a lightweight mechanism for describing properties of data structures, and applied groups to checking data sharing strategies in several multithreaded C programs. An important feature of our system is the group cast, which allows the programmer to describe where in the program the properties of a group change. For instance, in many of the C programs we examined, data structures switch between stages where they are protected by locks and stages where they are private to a single thread. We have proved the soundness of groups, and implemented Shoal, our group-based sharing checker as an extension to the earlier SharC system. We are able to check the correctness of sharing in our benchmarks with reasonable overhead (6%–42%) and a reasonable annotation burden.

Our formalism for groups is not strongly tied to the concept of a sharing mode. In the future, we plan to investigate further uses of groups, including memory management and tracking user-specified properties like tainting. We also wish to continue improving Shoal, to reduce concurrent reference counting overhead, to be able to describe and check more kinds of sharing, such as double-checked locking [26] and to be able to check more kinds of sharing statically, rather than, like `barrier`, dynamically.

A. Soundness Proof

Our basic approach is to prove, by induction over the operational semantic steps that the following two properties hold at all times:

- The statements in all threads are well-typed. In particular, our typing rules require that runtime checks necessary before performing an assignment either hold or are yet to be performed (Section A.2).
- The memory and thread environments are consistent (Section A.3).

$\Gamma \vdash s \Rightarrow s'$	In environment Γ statement s compiles to s', which is identical to s except for added runtime checks.

(SEQ)
$$\frac{\Gamma \vdash s_1 \Rightarrow s_1' \qquad \Gamma \vdash s_2 \Rightarrow s_2'}{\Gamma \vdash s_1; s_2 \Rightarrow s_1'; s_2'}$$

(SPAWN)
$$\frac{f \in F}{\Gamma \vdash \text{spawn } f() \Rightarrow \text{spawn } f()}$$

(LOCK)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash \text{lock } x \Rightarrow \text{lock } x}$$

(UNLOCK)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash \text{unlock } x \Rightarrow \text{unlock } x}$$

Figure 12. Elided rules. F is the set of all thread functions in the program.

From these properties it is easy to prove that private cells are only accessed by their owner and that locked cells are only accessed when the lock is held.

The proof consists of a few general-usage lemmas (Section A.4), proofs that single-thread steps, thread creation and thread destruction preserve the two properties (Sections A.5 through A.7), and a final section that puts all the pieces together to prove that the statically declared sharing modes are respected (Section A.8).

A.1 Preliminaries

We use *address* to refer to an element of the domain of a memory, and note that 0 is never a valid address ($0 \notin \text{dom}(M)$). We use *cell* to refer to an actual memory element $M(a)$. An *lvalue* is an expression that denotes a field of a particular cell.

We use letters $a-d$ to refer to addresses, $i-n$ to refer to integers and all other letters to refer to identifiers. Normally, letters $f-h$ refer to fields, $x-z$ to variables, and $t-v$ to structure type names.

A.2 Runtime Typing

To show that our operational semantics preserve types, we show that programs remain well-typed at all points during execution. To do this, we need typing rules that enforce the presence of runtime checks rather than add them, and handle the runtime-only statements (**skip**, **wait**). These typing rules are given in Figure 13 and are derived from the typing judgments of Figure 8, completed by the rules in Figure 12.

$M, id, r \models s$ checks that s is a well-typed statement of thread id in memory M. Runtime checks are special: if r is false, the checks for an assignment must all be present in its when clause. If r is true, then a prefix of the necessary checks can instead hold in M and be omitted from the when clause. For soundness, we must only check an assignemnt with r true when it is the first statement of a thread: this is enforced by passing false for r when checking s_2 in SEQ-R. For convenience, we write $M, id \models s$ to stand for $M, id, \text{true} \models s$.

A.3 Consistency

To ensure that programs remain type-safe, we need to know that types and owners in the memory are consistent. Furthermore, to avoid group casts changing the types of variables, we must assert that the memory cell containing a thread's environment is unaddressable.

Definition 1. *Memory consistency.* M is *consistent* with threads $id_1, \ldots, id_n$, written $id_1, \ldots, id_n \models M$, if all thread identifiers are distinct, cell id_i is unaddressed and owned by id_i, and types and owners are consistent when an lvalue refers to a cell. Formally for all threads id_i

$$M, id \models \ell : \tau, \omega$$

In memory M and thread id, ℓ is a well-typed expression with type τ assuming ω holds.

(NAME-R)
$$\frac{M_\rho(id.x) = m{<}y{>}\ t}{M, id \models x : m{<}y{>}\ t, \epsilon}$$

(SAME FIELD-R)
$$\frac{M_\rho(id.x) = m{<}y{>}\ t \qquad T(t.f) = t'}{M, id \models x.f : m{<}y{>}\ t', m(x)}$$

(OTHER FIELD-R)
$$\frac{M_\rho(id.x) = m{<}y{>}\ t \qquad T(t.f) = m'{<}g{>}\ t'}{M, id \models x.f : m'{<}x.g{>}\ t', m(x)}$$

$M, id, r \models s$	In memory M and thread id, statement s is well-typed. If r is true, valid runtime conditions may be assumed.

(SEQ-R)
$$\frac{M, id, r \models s_1 \qquad M, id, \text{false} \models s_2}{M, id, r \models s_1; s_2}$$

(SPAWN-R)
$$\frac{}{M, id, r \models \text{spawn } f()}$$

(LOCK-R)
$$\frac{M_\rho(id.x) = \tau}{M, id, r \models \text{lock } x}$$

(UNLOCK-R)
$$\frac{M_\rho(id.x) = \tau}{M, id, r \models \text{unlock } x}$$

(DONE-R)
$$\frac{}{M, id, r \models \textbf{done}}$$

(SKIP-R)
$$\frac{}{M, id, r \models \textbf{skip}}$$

(NEW-R)
$$\frac{\begin{array}{c} M_\rho(id.x) = m{<}y{>}\ t \\ \mathrm{D}(M, id, x) = \omega_1, \ldots, \omega_n \qquad r \wedge \forall i \in 1..k-1 : M, id \models \omega_i \end{array}}{M, id, r \models x := \text{new when } \omega_k, \ldots, \omega_n}$$

(GCAST-R)
$$\frac{\begin{array}{c} x \neq z \qquad M_\rho(id.x) = m{<}y{>}\ t \qquad M_\rho(id.z) = m'{<}z{>}\ t \\ oneref(z), \mathrm{D}(M, id, x), \mathrm{D}(M, id, z) = \omega_1, \ldots, \omega_n \\ r \wedge \forall i \in 1..k-1 : M, id \models \omega_i \end{array}}{M, id, r \models x := \text{gcast } z \text{ when } \omega_k, \ldots, \omega_n}$$

(READ-R)
$$\frac{\begin{array}{c} M_\rho(id.x) = m{<}x'{>}\ t \qquad M, id \models \ell : m{<}\ell'{>}\ t, \omega \\ \omega, x'[\ell/x] = \ell', \mathrm{D}(M, id, x) = \omega_1, \ldots, \omega_n \\ r \wedge \forall i \in 1..k-1 : M, id \models \omega_i \end{array}}{M, id, r \models x := \ell \text{ when } \omega_k, \ldots, \omega_n}$$

(WRITE-R)
$$\frac{\begin{array}{c} M, id \models x.f : m{<}\ell{>}\ t, \omega \qquad M_\rho(id.y) = m{<}y'{>}\ t \\ \omega, \ell[y/x.f] = y', \mathrm{D}(M, id, x.f) = \omega_1, \ldots, \omega_n \\ r \wedge \forall i \in 1..k-1 : M, id \models \omega_i \end{array}}{M, id, r \models x.f := y \text{ when } \omega_k, \ldots, \omega_n}$$

Runtime checks for dependent-type assignments

$$\mathrm{D}(M, id, x) = \{y = \text{null} \mid x \neq y \wedge M_\rho(id.y) = m{<}x{>}\ t\}$$
$$\mathrm{D}(M, id, x.f) = \{x.g = \text{null} \mid M_\rho(id.xx) = m{<}y{>}\ t \wedge f \neq g \wedge t(g) = m{<}f{>}\ t\}$$

Figure 13. Runtime typing judgments.

1. $id_i \neq 0$, and $i \neq j \Rightarrow id_i \neq id_j$ (threads are distinct)
2. $M_\rho(id_i) = \texttt{private}{<}0{>}\ t$ (thread environments not in groups)
3. $M_o(id_i) = id_i$ (thread environments are thread-owned)
4. $\nexists a.f (M_v(a.f) = id_i)$ (thread environments are unaddressed)

and for all addresses a where $M_\rho(a) = m{<}b{>}\ t$ with $t = (f_1 : \phi_1, \ldots, f_n : \phi_n)$:

5. if $c = M_v(a.f_i) \neq 0$ then
   (a) if $\phi_i = v$ then $M_\rho(c) = m{<}b{>}\ v$
   (b) if $(\phi_i = \texttt{private}{<}f_j{>}\ v) \vee (\phi_i = v \wedge m = \texttt{private})$ then $M_o(c) = M_o(a)$
   (c) if $\phi_i = m'{<}f_j{>}\ u$ then $M_\rho(c) = m'{<}M_v(a.f_j){>}\ u$
6. if $\phi_i = \texttt{private}{<}f_j{>}\ u$ then $b = 0 \wedge m = \texttt{private}$

A.4 Basic Properties

LEMMA 1. *Thread sets and consistency.* If $id_1, \ldots, id_n \models M$ then $id \models M$ If $id_1, \ldots, id_n \models M$, $id' \models M$ and $\forall i.id' \neq id_i$ then $id_1, \ldots, id_n, id' \models M$

LEMMA 2. *Private accesses by owning thread only.* Assume

$$id \models M \qquad M_\rho(a) = \texttt{private}{<}b{>}\ t$$

If $M, id : s \xrightarrow{s} M'$ causes thread id to access (read or write) cell a then $M_o(a) = id$
Corollary: If $M_o(a) \neq id$ then $M'_v(a) = M_v(a)$
Proof: By inspection of the operational semantic rules. First, all accesses to $M_v(id)$ are safe as $id \models M$ implies $M_o(id) = id$. Second, all accesses to cells $a = M_v(id.x)$ (due to lvalue $x.f$) are safe as $M_\rho(a) = \texttt{private}{<}b{>}\ t$ implies $M_\rho(id.x) = \texttt{private}{<}y{>}\ t$, which itself implies that $M_o(a) = M_o(id) = id$ (both from $id \models M$).

LEMMA 3. *Locked accesses by locking thread only.* Assume

$$id \models M \qquad M, id \models s \qquad M_\rho(a) = \texttt{locked}{<}b{>}\ t$$

If $M, id : s \xrightarrow{s} M'$ causes thread id to access (read or write) cell a then $M_L(a) = id$
Corollary: If $M_L(a) \neq id$ then $M'_v(a) = M_v(a)$
Proof: By inspection of the operational semantic rules. First, all accesses to $M_v(id)$ are safe as $id \models M$ implies $M_\rho(id) = \texttt{private}{<}c{>}\ u$. Second, if thread id accesses cell $a = M_v(id.x)$ due to lvalue $x.f$, then $M_\rho(a) = \texttt{locked}{<}b{>}\ t$ implies $M_\rho(id.x) = \texttt{locked}{<}y{>}\ t$ (from $id \models M$). Inspection of the READ-R and WRITE-R typing rules (which check the only assignments that can access a) shows that $\omega_1 = \texttt{locked}(x)$, i.e. $\texttt{locked}(x)$ is checked first. Thus when the assignments' subsequent runtime checks, or the assignmemt itself access a, $M, id \models \texttt{locked}(x)$ and hence $M_L(a) = id$.

LEMMA 4. *Lvalue types are respected.* If

$$id \models M \qquad M, id \models \ell : m{<}\ell'{>}\ t, \omega$$
$$lval(M, id, \ell) = a.f \qquad c = M_v(a.f) \neq 0 \qquad lval(M, id, \ell') = b.g$$

then

$$M_\rho(c) = m{<}M_v(b.g){>}\ t \qquad m = \texttt{private} \Rightarrow M_o(c) = id$$

Proof: Follows from the definition of memory consistency, and clause 6 of $id \models M$.

LEMMA 5. *Dependent types safe under assignment.* If

$$id \models M \qquad \mathrm{D}(M, id, \ell) = \omega_1, \ldots, \omega_n \qquad \forall i.M, id \models \omega_i$$
$$lval(M, id, \ell) = a.f \qquad M' = M[a.f \xrightarrow{v} b]$$

then $\forall g \neq f. M'_\rho(a.g) = m{<}h{>}\ t \wedge M'_v(a.g) \neq 0 \Rightarrow M'_\rho(M'_v(a.g)) = m{<}M'_v(a.h){>}\ t$ (clause 5c preserved for all fields but f)
Proof: Trivial for all cases except $h = f$. When $h = f$, $a.g = \text{null} \in \mathrm{D}(M, a, f)$ and hence $M_v(a.g) = 0 = M'_v(a.g)$, so the lemma holds.

Lemma 6. *Thread steps preserve local variable types* If

$$id \models M \qquad M, id : s \xrightarrow{s} M'$$

then $M'_\rho(id) = M_\rho(id)$
Proof: Only *gcast* changes existing cell types, and only for cells whose group is non-zero. As $M_\rho(id) = \texttt{private}\texttt{<}0\texttt{>}\ t$, $M'_\rho(id) = M_\rho(id)$.

Lemma 7. *Other-thread steps preserve local variable values.* If

$$id, id' \models M \qquad M, id' : s \xrightarrow{s} M'$$

then $M'_v(id.x) = M_v(id.x)$
Proof: $M_\rho(id) = \texttt{private}\texttt{<}0\texttt{>}\ t$ and $M_o(id) = id$, so by Lemma 2 $M'_v(id.x) = M_v(id.x)$.

Lemma 8. *Prototype thread preservation.* If

$$\begin{array}{c} id_f \models M \qquad \models M[id_f \xrightarrow{o} 0] \\ id_f \models M' \qquad M'_\rho(id_f) = M_\rho(id_f) \qquad M'_v(id_f) = M_v(id_f) \end{array}$$

then $\models M'[id_f \xrightarrow{o} 0]$
Proof: Let $M'' = M'[id_f \xrightarrow{o} 0]$. Consider any lvalue $a.g$ in M'' with $c = M''_v(a.g) = M'_v(a.g)$ and $\phi = M''_\rho(a.g) = M'_\rho(a.g)$. If $c = 0$ then clause 5 holds. Otherwise, $id_f \models M'$ implies $c \neq id_f$, so $M''_\rho(c) = M'_\rho(c)$ and $M''_o(c) = mo'(c)$. For $a \neq id_f$, $M''_o(a) = M'_o(a)$, so $id_f \models M'$ implies that clause 5 holds for lvalue $a.g$. If $a = id_f$ $c = M_v(id_f.g)$ and $\phi = M_\rho(id_f.g)$. If $\phi = v \vee \phi = \texttt{private}\texttt{<}g\texttt{>}\ t$ then $M_o(c) = M_o(id_f) = id_f$ and $M_o(c) = M[id_f \xrightarrow{o} 0]_o(id_f) = 0$, a contradiction. Thus $\phi = m\texttt{<}h\texttt{>}\ v$ with $m \neq \texttt{private}$ and clauses 5a and 5b hold on M''. Clause 5c holds because $M''_\rho(c) = M'_\rho(c) = m\texttt{<}M'_v(id_f.h)\texttt{>}\ v = m\texttt{<}M''_v(id_f.h)\texttt{>}\ v$. Clause 6 holds because $M''_\rho = M'_\rho$.

A.5 Preserving Consistency

Lemma 9. *Single-thread steps preserve memory consistency.* If

$$M, id \models s \qquad M, id : s \xrightarrow{s} M'$$

then $id, id' \models M \Rightarrow id, id' \models M'$
Proof: We prove the various aspects of consistency independently, each time by case inspection of the operational semantics' steps.

C1 Remains valid.

C2,3 Trivial for all steps except gcast. We know that $M_\rho(id) = \texttt{private}\texttt{<}0\texttt{>}\ t$, so $M'_\rho(id) = M_\rho(id)$ and $M'_o(id) = M_o(id)$ from the definition of *gcast*.

C4 By inspection, all steps that cause $M'_v(a.f) \neq M_v(a.f)$ define $M'_v(a.f)$ as 0 (null), $c \notin \text{dom}(M)$ (new) or an existing value $M_v(b.g)$. Thus $id \neq 0$, $id \in \text{dom}(M)$ and $\nexists a.f(M_v(a.f) = id)$ imply that $\nexists a.f(M'_v(a.f) = id)$.

C5 Trivial except for assignment steps.

- $x := \ell$, with $lval(M, id, \ell) = a.f$, $c = M_v(a.f)$.
  By Lemma 5, clause 5c is preserved for all variables dependent on x. If $c = 0$ the remaining conditions are trivially true. Otherwise, we must show clauses 5a-c holds for lvalue $id.x$. By assumption
  $$\begin{array}{cc} M_\rho(id.x) = m\texttt{<}x'\texttt{>}\ t & M, id \models \ell : m\texttt{<}\ell'\texttt{>}\ t, \omega \\ M, id \models x'[\ell/x] = \ell' & lval(M, id, \ell') = b.g \end{array}$$
  Clause 5a is trivial.
  Clause 5b follows directly from Lemma 4 applied to lvalue ℓ ($m = \texttt{private} \Rightarrow M_o(c) = id$).
  By Lemma 4, $M'_\rho(c) = M_\rho(c) = m\texttt{<}M_v(b.g)\texttt{>}\ t$. So, for clause 5c, we must verify that $M'_v(id.x') = M_v(b.g)$. If $x' = x$, $M'_v(id.x') = c$, and $M, id \models \ell = \ell'$ implies $c = M_v(b.g)$. If $x' \neq x$, $M'_v(id.x') = M_v(id.x')$ and $M, id \models x' = \ell'$ implies $M_v(id.x') = M_v(b.g)$.
- $x.f := y$, with $a = M_v(id.x) \neq 0$, $c = M_v(id.y)$.
  By Lemma 5, clause 5c is preserved for all variables dependent on $x.f$. If $c = 0$ the remaining conditions are trivially true. Otherwise, we must show clauses 5a-c holds for lvalue $a.f$. By assumption
  $$\begin{array}{cc} M_\rho(id.x) = m'\texttt{<}x'\texttt{>}\ v & M_\rho(id.y) = m\texttt{<}y'\texttt{>}\ t \\ M, id \models x.f : m\texttt{<}\ell\texttt{>}\ t, \omega & M, id \models \ell[y/x.f] = y' \\ \multicolumn{2}{c}{lval(M, id, \ell) = b.g} \end{array}$$
  Clause 5b follows from Lemma 4 applied to lvalue y ($m = \texttt{private} \Rightarrow M_o(c) = id$).
  First we consider the case where $T(t.f) = u$. Then $m = m'$, $\ell = x'$ and $M, id \models x' = y'$. From Lemma 4, $M'_\rho(c) = M_\rho(c) = m\texttt{<}M_v(id.y')\texttt{>}\ t$. From clause 4 of memory consistency we can conclude that $a \neq id$, hence $M'_v(id.x') = M_v(id.x') = M_v(id.y')$ and clause 5a holds. Clause 5c is trivial.
  Next we consider the case where $T(t.f) = m\texttt{<}f\texttt{>}\ t$. Then $\ell = x.f$ and $M, id \models y = y'$. From Lemma 4, $M'_\rho(c) = M_\rho(c) = m\texttt{<}M_v(id.y')\texttt{>}\ t$. As $M'_v(a.f) = M_v(id.y) = M_v(id.y')$, clause 5c holds. Clause 5a is trivial.
  Finally we consider the case where $T(t.f) = m\texttt{<}g\texttt{>}\ t$, $f \neq g$. Then $\ell = x.g$ and $M, id \models x.g = y'$. From Lemma 4, $M'_\rho(c) = M_\rho(c) = m\texttt{<}M_v(id.y')\texttt{>}\ t$. As $M'_v(a.g) = M_v(a.g) = M_v(id.y')$, clause 5c holds. Clause 5a is trivial.
- $x :=$ new, with $a = M'_v(id.x)$
  By Lemma 5, clause 5c is preserved for all variables dependent on x. We must show clauses 5a-c holds for lvalue $id.x$. Clauses 5a and 5b hold trivially. Clause 5c holds by construction of the $M'_\rho(a)$. Finally, clause 5 holds for the fields of cell a as they are all null.
- $x :=$ gcast z, with $M_\rho(id.x) = m\texttt{<}y\texttt{>}\ t$, $M_\rho(id.z) = m'\texttt{<}z\texttt{>}\ t$, $c = M_v(id.z)$, $M, id \models \mathit{oneref}(z)$.
  Let $M'' = M[id.x \xrightarrow{v} c, id.z \xrightarrow{v} 0]$. By two applications of Lemma 5, clause 5c is preserved in M'' for all variables dependent on x and z. If $c = 0$, $M' = M''$ and clause 5 holds. We verify that clause 5 holds for all addresses a when $c \neq 0$:
  Cells outside the group being cast. If $M_\rho(a) = n\texttt{<}b\texttt{>}\ u$ with $b \neq c$ and $a \neq id$, then $M(a) = M'(a)$. Consider a field f of a such that $b = M'_v(a.f) \neq 0$. If $M'_\rho(a.f) = v$ then $M_\rho(b) = n\texttt{<}b\texttt{>}\ v$. As $b \neq c$, $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$ so clauses 5a and 5b hold. Clause 5c holds trivially (no other fields can be dependent on f). If $M'_\rho(a.f) = n'\texttt{<}g\texttt{>}\ v$, then $M_\rho(b) = n'\texttt{<}M_v(a.g)\texttt{>}\ v$. $\mathit{oneref}(z)$ implies that $M_v(a.g) \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.
  Cells inside the group being cast. If $M_\rho(a) = n\texttt{<}c\texttt{>}\ u$ (this implies $a \neq id$) then $M'_\rho(a) = m\texttt{<}d\texttt{>}\ u$, $M'_o(a) = id$, and $\forall f.M'_v(a.f) = M_v(a.f)$ where $d = M''_v(id.y) = M'_v(id.y)$. Consider a field f of a such that $b = M'_v(a.f) \neq 0$. If $M'_\rho(a.f) = v = M_\rho(a.f)$ then $M_\rho(b) = n\texttt{<}c\texttt{>}\ v$, so $M'_\rho(b) = m\texttt{<}d\texttt{>}\ v$ and $M'_o(b) = id$ so clauses 5a and 5b hold. Clause 5c holds trivially (no other fields can be dependent on f). If $M'_\rho(a.f) = n'\texttt{<}g\texttt{>}\ v = M_\rho(a.f)$, then $M_\rho(b) = n'\texttt{<}M_v(a.g)\texttt{>}\ v$. $\mathit{oneref}(z)$ implies that $M_v(a.g) \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.
  Local variables of id. Consider a local variable x' of thread id with type $n\texttt{<}y'\texttt{>}\ u$.
  - $x' = z$: Clauses 5a-5c hold trivially.
  - $x' = x$: $M_\rho(c) = m'\texttt{<}c\texttt{>}\ t$, so $M'_\rho(c) = m\texttt{<}d\texttt{>}\ t$ where $d = M''_v(id.y) = M'_v(id.y)$ so clauses 5a-5c hold.
  - $x' \neq x, x' \neq z$: if $b = M'_v(id.x') = M_v(id.x') \neq 0$ then $M_\rho(b) = n\texttt{<}M_v(id.y')\texttt{>}\ u$. If $y' = z$, then $M, id \models x' =$

null, a contradiction. Otherwise *oneref*(z) implies that $M_v(id.y') \neq c$, thus $M'_\rho(b) = M_\rho(b)$ and $M'_o(b) = M_o(b)$. Thus clauses 5a-5c hold.

C6 Trivial except for new and gcast steps, as types are unchanged. For new, the clause holds because STRUCTDEF prohibits explicit `private` fields within structures (this could be relaxed to a weaker rule matching the requirements of clause 6).
For gcast, consider an lvalue $a.f$ with $M'_\rho(a) = m'\text{<}c\text{>}\ t$, $M_\rho(a) = m\text{<}b\text{>}\ t$, $M'_\rho(a.f) = \texttt{private<}g\texttt{>}\ t = M_\rho(a.f)$. $\models M$ implies $b = 0$ and $m = \texttt{private}$, so *gcast* will leave cell a unchanged, i.e. $c = b = 0$ and $m' = m = \texttt{private}$, so clause 6 holds for M'.

A.6 Preserving Type Safety

LEMMA 10. *Same-thread steps preserve type safety* If

$$id \models M \qquad M, id : s \xrightarrow{s} M'$$

then $M, id \models s; s' \Rightarrow M', id \models s'$
Proof: From Lemma 6, $M'_\rho(id) = M_\rho(id)$, so $M'_\rho(id.x) = M_\rho(id.x)$ for all variables x of thread id. An inspection of the rules show that $M, id, \text{false} \models s'$ depends only on $M_\rho(id)$, and thus $M', id, \text{false} \models s'$, so $M', id \models s'$.

LEMMA 11. *Other-thread steps preserve type safety* If

$$id, id' \models M \qquad M, id' : s' \xrightarrow{s} M'$$

then $M, id \models s \Rightarrow M', id \models s$
Proof: From Lemma 6, $M'_\rho(id) = M_\rho(id)$, so $M'_\rho(id.x) = M_\rho(id.x)$ for all variables x of thread id. Thus, to show that $M', id \models s$ it suffices to show that already-executed runtime checks are preserved, i.e. $M, id \models \omega \Rightarrow M', id \models \omega$ for all ω that are required by $M, id \models s$. We analyse each check independently:

- `private`(x) always holds.
- `locked`(x): By Lemma 7, $a = M'_v(id.x) = M_v(id.x)$, and by assumption $M_L(a) = id$. The only step that could cause $M'_L(a) \neq M_L(a)$ is unlock y with $M_v(id'.y) = a$. However, this can only be executed by id' if $M_L(a) = id'$.
- ℓ = null: If $\ell = x$, by Lemma 7, $M'_v(id.x) = M_v(id.x)$ so $M', id \models \ell = \text{null}$. If $\ell = x.f$, by Lemma 7, $a = M'_v(id.x) = M_v(id.x)$. If $M_\rho(a) = \texttt{private<}b\texttt{>}\ t$, then, by Lemma 2, $M'_v(a.f) = M_v(a.f)$ so $M', id \models \ell = \text{null}$. If $M_\rho(a) = \texttt{locked<}b\texttt{>}\ t$ then from $id \models M$ we know that $M_\rho(id.x) = \texttt{locked<}y\texttt{>}\ t$. By inspection of the rules that depend on $\ell = \text{null}$, we note that $M, id \models \texttt{locked}(x)$ must hold, i.e. $M_L(a) = id$. Hence, by Lemma 3, $M'_v(a.f) = M_v(a.f)$ so $M', id \models \ell = \text{null}$
- $\ell = \ell'$: The same logic as for ℓ = null applies.
- *oneref*(x): We know that $id.x$ is the sole lvalue referencing $a = M_v(id.x)$. Only assignment statements by id' could cause this to change. $y = z$, null, new and gcast assignments cannot create or destroy references to a as they modify local variables of $id' \neq id$. $id \models M$ implies that no lvalue references id, so $y.f := z$ cannot modify $id.x$ and similarly that $y := z.f$ cannot create an extra reference to a. Thus $M', id \models$ *oneref*(x).

A.7 Thread Creation and Destruction

These two lemmas show that thread creation and destruction preserve type safety, runtime checks and memory and environmental consistency.

LEMMA 12. *Thread creation.* If

$$id_f, id \models M \qquad M, id_f \models \textbf{wait}; s_f \qquad M, id \models s$$
$$\models M[id_f \xrightarrow{o} 0] \qquad id' = \max(\text{dom}(M)) + 1$$
$$M' = extend(M, id', id', M_\rho(id_f)) \qquad M'' = M'[id' \xrightarrow{v} M_v(id_f)]$$

then

$$id_f, id, id' \models M''$$
$$M'', id_f \models \textbf{wait}; s_f \qquad M'', id \models s \qquad M'', id' \models s_f$$

Proof: $id_f, id, id' \models M'$ is trivial. Verifying $id_f, id, id' \models M''$ only requires checking that clause 5 holds for all lvalues $id'.y$. Consiser $a = M''_v(id'.y) = M_v(id_f.y)$, $M''_\rho(id'.y) = M_\rho(id_f.y) = \phi$. If $a = 0$, then clause 5 holds. Consider $a \neq 0$. If $\phi = v \vee \phi = \texttt{private<}z\texttt{>}\ t$ then $M_o(a) = M_o(id_f)$ and $M_o(a) = M[id_f \xrightarrow{o} 0]_v(id_f) = 0$, a contradiction. Thus $\phi = m\text{<}z\text{>}\ t$ with $m \neq \texttt{private}$ and clauses 5a and 5b hold on M''. Clause 5c holds because $M''_\rho(a) = M_\rho(a) = m\text{<}M_v(id_f.z)\text{>}\ t = m\text{<}M''_v(id'.z)\text{>}\ t$ (note that $a \neq id'$).

$M'', id_f \models \textbf{wait}; s_f$ follows from the fact that $M''_\rho(id_f) = M_\rho(id_f)$, and $M'', id' \models s_f$ follows from the fact that $M''_\rho(id') = M_\rho(id_f)$. Finally, $M'', id \models s$ follows from $M, id \models s$ and the fact that $M, id \models \omega \Rightarrow M'', id \models \omega$: the only interesting case is *oneref*(x). If $id.x$ is the only reference to cell $a = M_v(id.x)$ then $M_v(id_f.y) \neq a$, so M'' cannot invalidate *oneref*(x).

LEMMA 13. *Thread destruction.* If

$$id, id' \models M \qquad M, id \models s \qquad M' = M \backslash id'$$

then $id \models M'$ and $M', id \models s$.
Proof: We first show $id \models M'$. Clauses 1-4 follow directly from $id, id' \models M$ and $id \neq id'$. $id, id' \models M$ implies $\nexists a.f (M_v(a.f) = id')$, so clause 5 remains valid for all addresses $b \in \text{dom}(M')$. Clause 6 remains valid as $M'_\rho(b) = M_\rho(b)$ for all addresses $b \in \text{dom}(M')$.

As with thread creation, we note that $M, id \models \omega \Rightarrow M'', id \models \omega$ (easily verified by examing each kind of runtime check, noting that $M_v(id.x) \neq id'$). Furtheremore, $M'_\rho(id) = M_\rho(id)$ so $M', id \models s$.

A.8 Soundness

THEOREM 4. *Soundness.* Let P be a program with $\vdash P \Rightarrow P'$ and M_0, S_0 be the initial memory and threads for program P' with starting thread main shown in Figure 14. Let $M, \{(id_1, s_1), \ldots, (id_n, s_n)\}$ be a state reachable in i steps from M_0, S_0. Then

$$id_1, \ldots, id_n \models M \qquad M, id_i \models s_i$$

Proof: We prove the slightly stronger result that the state M, S after step i satisfies

$$S = \{(id_{f_1}, \textbf{wait}; s_{f_1}), \ldots, (id_{f_n}, \textbf{wait}; s_{f_n}), (id_1, s_1), \ldots, (id_n, s_n)\}$$
$$id_{f_1}, \ldots, id_{f_n}, id_1, \ldots, id_n \models M \qquad \models M[id_{f_i} \xrightarrow{o} 0]$$
$$M, id_{f_i} \models \textbf{wait}; s_{f_i} \qquad M, id_i \models s_i$$

where the f_i's are the threads declared in P'.

The proof proceeds by induction over the steps of the operational semantics.

By construction, M_0, S_0 satisfies the induction hypothesis.

Assume M, S satisfies the induction hypothesis, and $M, S \rightarrow M', S'$.

First, we handle the prototype threads id_{f_i}. Note that there are no thread step transitions from statements of the form **wait**; s. Thus, $(id_{f_i}, \textbf{wait}; s_{f_i}) \in S'$. Also $M'_\rho(id_{f_i}). = M_\rho(id_{f_i})$ and $M'_v(id_{f_i}) = M_v(id_{f_i})$ (Lemmas 6 and 7), so, by Lemma 8,

$$(id_f \models M') \Rightarrow (\models M'[id_f \xrightarrow{o} 0])$$

To complete the induction we thus only need to show

$$\forall (id, s) \in S' (id \models M') \wedge (M', id \models s)$$

We proceed by analysing each kind of state transition.

- Simple statement:

$$\frac{M, id : s_1 \xrightarrow{s} M'}{M, \{(id, s_1; s_2)\} \oplus S \rightarrow M', \{(id, s_2)\} \cup S}$$

$$\frac{\begin{array}{c}\vdash P \Rightarrow P' \qquad 0 \notin \mathrm{dom}(M) \qquad g > 0 \qquad \mathrm{dom}(M) = \{g\} \oplus (\bigoplus_{f()\{\ldots\}\in P'} \{id_f\}) \oplus (\bigoplus_{x:\tau\in P'} \{o_x\}) \qquad S = \{(id_f, \textbf{wait}; s) | f()\{\ldots; s\} \in P'\} \\ M(g) = (\texttt{private<0>}\ t_G, g, 0, v_G) \qquad x : \tau \in P' \Rightarrow t_G(x) = \tau \wedge v_G(x) = o_x \wedge M(o_x) = (rtype(M, g, \tau), g, 0, \lambda f.0) \\ f()\{\ldots x : \tau \ldots\} \in P' \Rightarrow M(id_f) = (\texttt{private<0>}\ t_f, id_f, 0, v_f) \wedge t_f(x) = \tau \wedge v_f(x) = 0 \wedge (y : \tau \in P' \Rightarrow t_f(y) = \tau \wedge v_f(y) = o_y)\end{array}}{M, S \oplus \{(g, \text{spawn main}; \textbf{done})\}}$$

Figure 14. Initial State

Let $(id', s') \in S$. By induction, $id, id' \models M$, $M, id \models s_1; s_2$ and $M, id' \models s'$. By Lemma 10 $M', id \models s_2$, by Lemma 11 $M', id' \models s'$ and by Lemma 9, $id, id' \models M'$. Lemma 1 completes the induction for this case.

- Runtime check:

$$\frac{M, id \models \omega_1 \qquad \Omega = \omega_2, \ldots, \omega_n}{M, \{(id, \ell := e \text{ when } \omega_1, \Omega; s)\} \oplus S \rightarrow M, \{(id, \ell := e \text{ when } \Omega; s)\} \cup S}$$

We only need to verify that

$$M, id \models id, \ell := e \text{ when } \omega_2 \ldots, \omega_n; s$$

Examination of the assignment rules shows that this holds because $M, id \models \omega_1$.

- Thread creation: The induction follows from Lemmas 12 and 1.
- Thread destruction: The induction follows from Lemmas 13 and 1.

THEOREM 5. *Safety of `private` accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{private<}b\texttt{>}\ t$ then $M_o(a) = id$.
Proof: From Theorem 4 and Lemma 2.

THEOREM 6. *Safety of `locked` accesses.* During the execution of a program P' such that $\vdash P \Rightarrow P'$, if thread id writes to cell a with type $M_\rho(a) = \texttt{locked<}b\texttt{>}\ t$ then $M_L(a) = id$.
Proof: From Theorem 4 and Lemma 3.

References

[1] ANDERSON, Z., GAY, D., ENNALS, R., AND BREWER, E. SharC: checking data sharing strategies for multithreaded C. In *PLDI'08*, pp. 149–158.

[2] BARNES, J., AND HUT, P. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature 324* (Dec. 1986), 446–449.

[3] BOYAPATI, C. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT.

[4] BOYAPATI, C., LEE, R., AND RINARD, M. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02*, pp. 211–230.

[5] BOYAPATI, C., LISKOV, B., AND SHRIRA, L. Ownership types for object encapsulation. In *OOPSLA'03*, pp. 213–223.

[6] BOYAPATI, C., SALCIANU, A., BEEBEE, JR., W., AND RINARD, M. Ownership types for safe region-based memory management in Real-Time Java. In *PLDI'03*, pp. 324–337.

[7] CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARAN, M. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI'02*, pp. 258–269.

[8] CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. Dependent types for low-level programming. In *ESOP'07*.

[9] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *POPL'99*, pp. 262–275.

[10] DEVELOPERS.SUN.COM. LockLint - static data race and deadlock detection tool for C. http://developers.sun.com/solaris/articles/locklint.html.

[11] ELMAS, T., QADEER, S., AND TASIRAN, S. Goldilocks: a race and transaction-aware Java runtime. In *PLDI'07*, pp. 245–255.

[12] ENGLER, D., AND ASHCRAFT, K. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP'03*, pp. 237–252.

[13] FLANAGAN, C., AND FREUND, S. N. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL'04*, pp. 256–267.

[14] FLANAGAN, C., FREUND, S. N., AND LIFSHIN, M. Type inference for atomicity. In *TLDI'05*, pp. 47–58.

[15] GAY, D., AND AIKEN, A. Language support for regions. In *PLDI'01*, pp. 70–80.

[16] GREENHOUSE, A., HALLORAN, T. J., AND SCHERLIS, W. L. Observations on the assured evolution of concurrent Java programs. *Sci. Comput. Program. 58*, 3 (2005), 384–411.

[17] GROSSMAN, D. Type-safe multithreading in Cyclone. In *TLDI'03*.

[18] KRISHNAMURTHY, A., CULLER, D. E., DUSSEAU, A., GOLDSTEIN, S. C., LUMETTA, S., VON EICKEN, T., AND YELICK, K. Parallel Programming in Split-C. In *SUPERCOM'93*, pp. 262–273.

[19] MCCLOSKEY, B., ZHOU, F., GAY, D., AND BREWER, E. Autolocker: synchronization inference for atomic sections. In *POPL'06*, pp. 346–358.

[20] NAIK, M., AND AIKEN, A. Conditional must not aliasing for static race detection. In *PLDI'07*, pp. 327–338.

[21] NECULA, G. C., MCPEAK, S., AND WEIMER, W. CIL: Intermediate language and tools for the analysis of C programs. In *CC'04*, pp. 213–228. http://cil.sourceforge.net/.

[22] PRATIKAKIS, P., FOSTER, J. S., AND HICKS, M. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI'06*, pp. 320–331.

[23] RAYSIDE, D., MENDEL, L., AND JACKSON, D. A dynamic analysis for revealing object ownership and sharing. In *WODA'06*, pp. 57–64.

[24] SASTURKAR, A., AGARWAL, R., WANG, L., AND STOLLER, S. D. Automated type-based analysis of data races and atomicity. In *PPoPP'05*, pp. 83–94.

[25] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP'97*, pp. 27–37.

[26] SCHMIDT, D., AND HARRISON, T. Double-Checked Locking. In Chapter 20 *of Pattern Languages of Program Design 3*, Addison-Wesley, ISBN 0201310112.

[27] TERAUCHI, T. Checking race freedom via linear programming. In *PLDI'08*, pp. 1–10.

[28] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. In *Information and Computation* (1977), vol. 132, pp. 109–176.

[29] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *SOSP'03*, pp. 268–281.

[30] VOUNG, J. W., JHALA, R., AND LERNER, S. RELAY: static race detection on millions of lines of code. In *ESEC-FSE'07*, pp. 205–214.

[31] WOO, S. C., OHARA, M., TORRIE, E., SHINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA'95*, pp. 24–36.

[32] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP'05*, pp. 221–234.